ECE 4750 Lab 1: Iterative Integer Multiplier

Basant Amro Khalil (bak94)

September 13, 2021

Section 1. Introduction

Multiplication is a critical component of many algorithms and programs to achieve their desired purposes. Many algorithms, especially the ones concerned with digital signal processing, need multiplication to perform many operations such that a huge amount of the time can be spent uselessly if there is no way of doing the multiplication efficiently. Now that this generation highly depends on technology and the Internet of Things (IoT), there will be more digital signal processing happening and so forth. Thus, our group felt the need to create an efficient way of running multiplication through creating an interactive integer multiplier that can support the multiplication operation efficiently. Consequently, this multiplier could be implemented in a multicore processor to be used with other modules.

Our group built 2 implementations of the interactive multiplier: the baseline design and the alternative design. The baseline design is a fixed latency, 34-35 cycle iterative multiplier that always takes the same number of cycles. The alternative design is a variable latency iterative multiplier that exploits properties of the input operands to reduce execution time that can be lower than the 34-35 cycles implemented for the baseline design such that the cycles run can be anywhere from 4 to 34 cycles. We exploited the potential structure of the operands a and b to decrease the number of cycles needed in the alternative design. The idea is to shift more than 1 bit one time in the presence of consecutive zeros. Even though we are not expecting much additional overhead on clock frequency in the alternative design, we are expecting a higher increase in area and energy. Implementing the tests that were created on the 2 designs show that the alternative design uses much fewer cycles than the baseline design: the alternative design used 35.12 cycles/multiplication vs the baseline design that used ranging from the lowest end which is 5 cycles/multiplication to the highest end of 31 cycles/multiplication. Therefore, using the evaluation results we can see that the alternative design showed better performance for all types of input datasets tested. The baseline design used the same number of cycles for all types of input datasets since it implements fixed latency design. This leads to wasting some cycles because of not exploiting the nature of the different types of input datasets, like the alternative design did. We can conclude then that the alternative design used much fewer number of cycles per multiplication from the sixth to about 90% of the number of cycles implemented for the baseline design.In terms of evaluation per different types of datasets, The alternative design showed the best use for the small inputs dataset since they have a bigger chance of having consecutive zeros that the design can exploit. In fact, the alternative design saved about 30.12 cycles per multiplication on a small input dataset compared to baseline design. It is also worth noting that the hardware use in the alternative design is higher compared to that of the baseline design. We should note that the alternative design has higher hardware costs than that of the baseline design causering both more area and energy consumption. Therefore, using lesser cycles per multiplication in the alternative design considering the higher hardware use in that design leads to the conclusion that the alternative design might be more effective and efficient than the baseline design for use with applications where technology constraints can accept this high energy and area consumption.

Our group expects the alternative design to be implemented in many more applications and algorithms for its efficiency while still being able to stay below the desired technology constraints for some applications, such as staying below certain energy consumption.

In this lab, we will implement the baseline and alternative designs, test the designs with certain tests, and perform an evaluation comparing the 2 implementations.

Section 2. Baseline Design

The baseline design is executed at the following interface: given a 64 bit req_msg that includes the operands, and other needed signals to follow the val/rdy interfaces, output the lower 32 bits result of the multiplication operation to the user. A picture is shown in figure 1 below that illustrates that concept. The algorithm we followed for executing the multiplication operation is explained below. The function should include 2 operands , call them a and b, as illustrated in the pseudocode in figure 2. If the least significant bit of the operand listed as the second , which in our case is b, is 1, then add the first operand , which is operand a , to the result. Then, shift the first operand, a, left by 1 bit and shift the second operand , b, right by 1 bit. This algorithm is the first one that comes to mind when we think of multiplication and is a good one to start with as a shift by 1 bit is expected to consume 1 cycle, leading to a theoretically simpler design. The pseudocode for this algorithm is displayed in figure 2 below.

Our code for the baseline design follows the logic implemented in the pseudocode. We originally have 3 registers, one for a, b , and the result. We implemented 2 shifters, one for logical left shifter and one for logical right shifter. We also implemented a multiplexer for adding to the result when the least significant bit of operand b is 1 as stated previously. Regarding the control logic, we did not implement it and the data flow as an inflexible implementation. The control module will send some control signals to the datapath module, changing the way data flows into registers. Some instances where this is implemented include when we accept new values for the operand registers, write or not-to-write to the result, and output an arithmetically executed result. For this to be implemented, the adatpath module will send the least significant bit of operand b to the control module such that it can select the needed data flow. In fact, the least significant bit of operand b is used to figure if we need to add to the sum of the result, which is also illustrated directly in the pseudocode. The datapath is illustrated in figure 3 below.

As illustrated in the pseudocode, we need to execute the for loop. Thus, we know from the way the for loop is executed in the pseudocode that this design will take at least 32 cycles to execute. As a matter of fact, our design takes from 34-35 cycles as we need one cycle to accept data and one cycle to establish that we are done. The finite state machine showing that design is illustrated in figure 4.

The baseline design uses patterns in the given data to execute. In the extreme case for which the second operand b is 0, the hardware will check every bit in b and use 32 cycles without adding to the result. The most efficient decision in this case would be to exploit that pattern to only one decision to not write to the result instead of 32 cycles.

We implemented modularity, which is a design technique that emphasizes separating the functionality of the program into independent, interchangeable modules, to exploit the use of structures, such as shifters, registers, and adders, through testing each one of them for correctness one by one. However, we could have also implemented hierarchy, which is an organizational structure where items are ranked according to levels of importance. It is shown in the diagram in figure 3 that each register has a multiplexer close to it to direct input. We instead could have designed a module where that structure is implemented. Then, we could have used this module 3 times. In that case, we would have been able to test unit by unit or in stages. In reality, we just tested all functionality one time. If we could have implemented the technique of reusing the module 3 times, we would have been able to earlier catch any RTL bugs earlier and have a more efficient code. However, we made sure to implement encapsulation(as relevant to the val/rdy interface) as a critical unit of our design.

Section 3. Alternative Design

The highest con of using the baseline design is the high latency associated with executing the multiplication operation. Here, our group will lay out some possible ideas for increasing performance with higher throughput and lower latency as possible. Theoretically, we can exploit caching and indirection, but we would not be allowed. Therefore, let us think about using pipelining for the multiplier. A key idea for the theoretical concept would be considering the way our operands execute to maximize performance. We will lay out 2 ways we can exploit the implementation of our operands. One way would be exploiting the presence of zeroes sequentially such that we could shift by more than 1 bit, if possible. Also, we will try to minimize the granularity associated with the number of zeros we can shift to powers of 2, minimizing some hardware costs and leading to higher performance compared to the baseline design.

Another consideration to keep in mind is , since the finite state machine dataflow is linear, we can use pipelining. We can implement different stages between the state changes and let the val/rdy interface decide the way the data flows between the pipelined stages. That might not be a great idea though because of the enormous hardware costs associated with it. We will need 34*3 = 102 registers, 34*4= 136 two ways multiplexer with 34 logical right and left shifters and 34 full adders. Therefore, that does not look like a great idea. It is also worth considering that this design will lead to a high throughput only if we need to do repeated multiplications.

Another possible design idea would be using single cycle processors. This design ,however, is likely to be on the critical path in an aggressive design. We will not be able to implement pipelining in that design so that design will not really add critical improvement. This design also seems not highly possible if we receive the instructions from a standard C program.

A better design idea would be taking advantage of the structure of our 32 bit operands. It is highly possible we will get a repeated sequence of zeroes of length x in the second operand. The algorithm we currently have will really not contribute to the result register, wasting x cycles. So, we can instead choose to shift x bits, saving x-1 cycles in the implementation. We will still need one cycle to figure out that there are actually x bits we can choose to not implement. The finite state machine for the alternative design is illustrated in figure 5 below. We will execute this logic using the caseZ

strategy. For instance, if we have many zeros in the least significant bits, we will increment the counter by x-1 and shift the needed amount. So, if we have all zeros in b, we will choose to shift 32 bits and execute only one cycle in the calc stage. Therefore, we will get the final total number of cycles as 3: one for IDLE, one for CALC, and one for DONE. If, however, we get the case where the second operand b contains all ones, we will execute the normal 34 cycles. That will lead to the range of the cycles we can take from 3 to about 34 or 35.

One of the pros of this idea is that our dataflow gets only a small change. We will just add a control signal to the shifters, and we will expand the status signal incorporated in the baseline design(which signaled the least significant bit of b) to the 32 bits of b as shown in figure 11. As shown for the data flow diagram of the alternative design in figure 11 that the hardware used in the alternative design is pretty similar to that used in the baseline design but there is some additional logic implemented like additional registers or muxes in the shifting logic. Therefore, we will see that the dataflow pieces of the alternative design are very similar to the one for baseline design, letting us only introduce minimal changes in a short time of coding to get higher efficiency. Also, instead of choosing to shift by x bits, we will shift by powers of 2. This will enable us to minimize the case numbers that we have to look at from 33 or 32 to 6 or 7. Our alternative design is theoretically better due to not wasting cycles for cases where shifting can be implemented. However, we need to mention that it will cause high area and energy consumption compared to the baseline design.

Section 4. Testing Strategy

We tested the baseline and alternative design implementations using the provided tests, and we also added to them some extra directed and random test cases. We made sure to design our directed and random test cases to consider different types of inputs, such as zeroes and ones, small and large numbers, positive and negative numbers, numbers with maksed lower/middle bits, and numbers featuring combinations of these types. We also made sure to make the inputs include different values from 0 to 32 bits(even though we know the width of all inputs is technically 32 bits.) In addition, we added test source/sink delays to the random tests to check for the val/rdy interface. We created a table below to show a summary of all the types of cases we checked for. First, the baseline design tested correctly with all these tests, but the alternative design did not pass some of them in contrast. We had to use line traces to ensure that enough granular testing is computed where outputs are generated as expected at the expected times.

After checking the errors we got for the alternative design, we were able to make the test cases pass for both the baseline and the alternative designs. Even though the test cases covered all the possible cases, we could have followed another path where we would consider the structure of the datapath and control modules separately with different individual tests to ensure that they are behaving the expected way. However, the test cases we created were enough to check for all possible conditions so there is no need to implement a different approach here.

Testing Cases	Testing Cases	Testing Cases
Combinations of multiplying zero, one, and negative one	Small positive numbers × small positive numbers	Small negative numbers × small positive numbers
Small negative numbers × small positive numbers	Small negative numbers × small negative numbers	Large positive numbers × large positive numbers
Large positive numbers × large negative numbers	Large negative numbers × large positive numbers	Large negative numbers × large negative numbers
Multiplying numbers with the low order bits masked off	Multiplying numbers with middle bits masked off	Sparse positive or negative numbers multiplied with spare positive or negative numbers
Multiplying dense positive or negative numbers with dense positive or negative numbers	Random small numbers multiplied with random small numbers	Random small numbers multiplied with random delays
Random large numbers multiplied by random small numbers	Random large numbers multiplied with random large numbers with random delays	Random small numbers multiplied with random small numbers with low bits masked off
Random small numbers multiplied	Random large numbers multiplied	

with random small numbers and low bits masked with random delays	with random large numbers with middle bits masked off	

Section 5. Evaluation

After we finished the baseline and alternative implementations, we tested the performance of both designs to compare them. We did that using the simulator harness which was created using the given dataset of numbers to create the total number of cycles and the average number of cycles per transaction for both the baseline and alternative designs. We also created eight different types of numbers to the simulator to compare the performance for the designs with different inputs: large numbers, low bits masked, middle bits masked, upper bits masked, large numbers and both are negative, large numbers and one is negative, small numbers and both are negative and small numbers and one is negative. We created a table below that contains the results we obtained for the different inputs implemented for both the baseline and the alternative design. We can conclude from the table that the baseline design illustrates the same performance for all different types of numbers here, which is expected since it is implementing fixed latency. In contrast, the alternative design is showing better performance compared to the baseline implementation for all input datasets here as it exploited the presence of consecutive zeros in inputs to minimize latency and maximize throughput. Some pictures of the program displaying the number of cycles are shown in figures 6 to 10 below.

It is worth noting that the alternative design showed better performance on the small input dataset compared to the large input dataset. That might be due to the fact that the large input dataset could contain less consecutive zeros and denser inputs that the design would be less efficient for. We can also see that the alternative design showed better performance for the masked numbers compared to the large input dataset due to the guaranteed existence of consecutive zeros. One idea we had but were not able to practice due to time was to exploit consecutive ones in negative numbers. We could have flipped the negative numbers to their positive equivalents by flipping those bits and then adding 1, then multiply the positive numbers instead and flip back at the end to negative results. By converting the negative numbers to their positive equivalents, we can exploit the alternative design's capability of showing better performance in relation to consecutive zeroes as it would not show better performance for consecutive ones instead. Those ideas could be implemented later to show more efficiency through minimizing latency and better improving performance, through exploiting concepts like parallelism and pipelining.

Thus, we can see that for the types of input dataset we tested below as shown in the table, the alternative design showed better performance for all of them. For input dataset like the small numbers and both numbers are negative, the use of the alternative design doesn't provide a much better performance compared to the baseline design with only about 4.12 cycles improvement. However, for inputs like the small input dataset, the use of the alternative design provides a huge improvement in performance compared to that or the baseline design with about 30.12 lesser cycles implemented in the alternative design. It should be noted too that the alternative design needs a higher hardware cost than that of the baseline design. The alternative design implements more muxes, adders and shifters than the baseline design does, costing more hardware costs, higher area and energy consumption. A design which requires a lot of logic and registers will require a lot of gates and take up more area on the chip compared to a design which uses less logic. Since more transistors mean more power consumption, so the amount of power consumed is going to depend on how much hardware the design uses. Therefore, for some applications that depend heavily on masked numbers or small and large inputs that are not negative, the implementation of alternative design would save many cycle times that the hardware costs, area, and energy costs can be covered. However, for applications that don't rely heavily on these input datasets or such, the use of the alternative design will not be a great improvement due to the hardware, area and energy costs outlined with it.

Design	Data Set: Small Inputs	Data Set: Large Inputs	Data set: Low Bits Masked	Data Set: Middle Bits masked	Data Set: Upper Bits Masked	Data Set: Large Numbers and Both Are Negative	Data Set: Large and One Number Is Negative	Data Set: Small Numbers and Both Are Negative	Data Set: Small and One Number Is Negative
Baseline	1756	1756	1756	1756	1756	1756	1756	1756	1756

design: Total cycles									
Baseline design: average cycles per multiplication	35.12	35.12	35.12	35.12	35.12	35.12	35.12	35.12	35.12
Alternative design: Total Cycles	263	931	512	550	560	931	926	1550	250
Alternative design: average cycles per multiplication	5.26	18.62	10.24	11	12	18.62	18.52	31.00	5.00

Section 6. Role and Task Table

Member 1 : Peter Wade	**Member 2: Lakshmi Aparna Bolla**	**Member 3: Basant Khalil**
RTL Design Engineer	RTL Verification Engineer	RTL Design Architect
Adding the needed resets and muxes for the baseline design according to the dataflow path module and pseudocode. Adding code to connect these modules together to create the desired result.	Helping with debugging some bugs in the main baseline code. For instance, reminding the lab members of how to figure out why the compiler is showing an error through using --tb=short command when testing. She also reminded the lab members of how to implement line tracing because we did not know where to find it in the tutorials.	Providing suggestions for the types of test cases that should be covered to test for all possible input datasets and corner cases, for which some tests would be: combinations of multiplying zero, one, and negative one, small negative numbers × small positive numbers, large positive numbers × large negative numbers, multiplying numbers with the low order bits masked off, multiplying dense positive or negative numbers with dense positive or negative numbers, random large numbers multiplied by random small numbers, random small numbers multiplied with random small numbers and low bits masked with random delays, small positive numbers × small positive numbers, small negative numbers × small negative numbers, large negative numbers × large positive numbers, multiplying
Providing suggestions for the RTL Verification Engineer for different possible ways to create a certain test case like providing suggestions to randomly generate a number densely packed with ones through possibly randomly selecting each bit and setting the weights to favor 1.	Creating various types of test cases for the alternative and baseline design including tests, such as for which some tests would be: combinations of multiplying zero, one, and negative one, small negative numbers × small positive numbers, large positive numbers × large negative numbers, multiplying numbers with the low	

Fixing a bug in the test cases that were pushed to verilog where he fixed some non ascii characters in the test case code that was preventing travis ci initially from showing that the code passes the test cases Contributing to the lab report through mentioning how the baseline design works. Also, contributing to the explanation that the alternative design will cost hardware costs and high energy and area consumption that need to be accounted for in comparison of performance. Explaining to a team member how to fix some errors that appeared when writing the provided commands by the lab member.	order bits masked off, multiplying dense positive or negative numbers with dense positive or negative numbers, random large numbers multiplied by random small numbers, random small numbers multiplied with random small numbers and low bits masked with random delays, small positive numbers × small positive numbers, small negative numbers × small negative numbers, large negative numbers × large positive numbers, multiplying numbers with middle bits masked off , and random small numbers multiplied with random small numbers. Contributing to the lab report through mentioning how the alternative design works Creating test cases for corner cases like cases where both numbers are 0 or negative.	numbers with middle bits masked off , and random small numbers multiplied with random small numbers. Creating finite state machine design for the alternative design and collaborating with other members to come up with a basic design for the alternative design and discussing ways to implement it exploiting the shared structure with the baseline design. Providing some suggestions of how the code will look like for the alternative design and discussing with other team members if that potential design would work Creating the evaluation table for comparing the performance for the alternative design versus the baseline design and contributing to the evaluation, testing and introduction portions of the lab report. Providing some suggestions to fix some of the errors that existed in the rtl code such as forgetting to implement reset in the state transitions portion of the code

Section 7. Referenced Figures

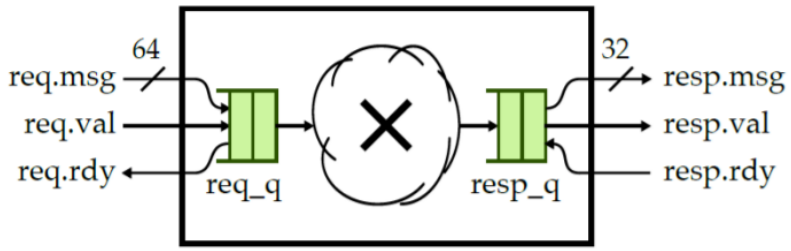


Figure 1: Functional-Level Implementation of Integer Multiplier - Input and output use latency-insensitive val/rdy interfaces. The input message includes two 32-bit operands; output message is a 32-bit result. Clock and reset signals are not shown in the figure.

```
def imul( a, b ):
  result = 0
  for i in range(32):
    if b & 0x1 == 1:
      result += a
    a = a << 1
    b = b >> 1
  return result
```

Figure 2: Iterative Multiplication Algorithm – Iteratively use shifts and subtractions to calculate the partial-products over time.

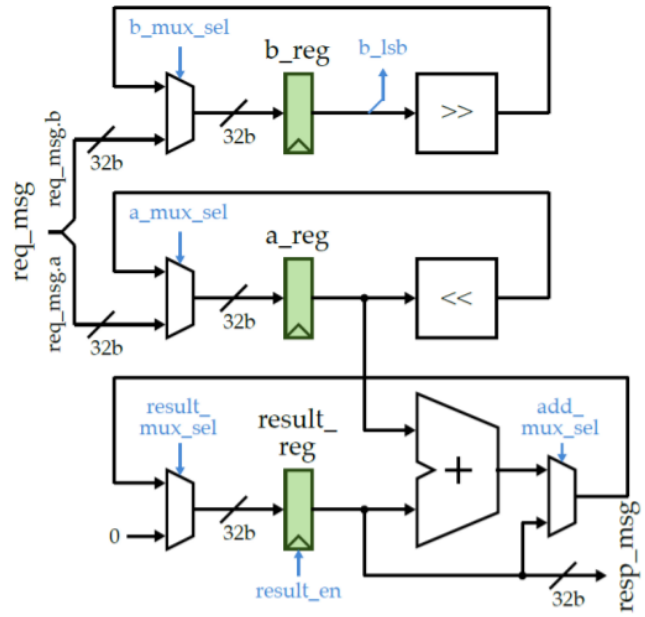


Figure 3: Datapath for Fixed-Latency Iterative Integer Multiplier - All datapath components are 32-bits wide. Shifters are constant one-bit shifters. We use registered inputs with a minimal of logic before the registers.

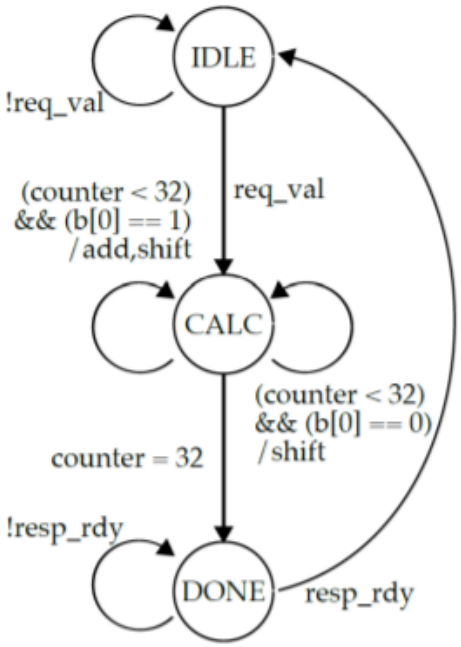
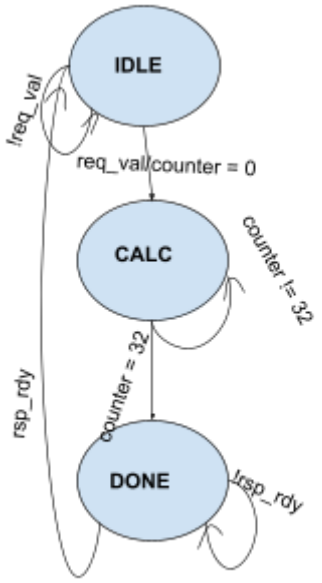


Figure 4: Control FSM for Fixed-Latency Iterative Integer Multiplier - Hybrid Moore/Mealy FSM with Mealy transitions in the CALC state.



,

Figure 5: alternative design finite state machine

```
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input small --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input large --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl alt --input small --stats
num_cycles         = 263
num_cycles_per_mul = 5.26
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl alt --input large --stats
num_cycles         = 931
num_cycles_per_mul = 18.62
ECE4750: ~/.../sim/build %
```

Figure 6: the program is displaying some information about the number of cycles for both the baseline and alternative design for small and large input datasets

```
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input mask_middle --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input mask_upper --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build %
```

Figure 7: the program displaying some information about the number of cycles for the baseline design for masked middle and upper bits

```
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl alt --input small --stats
num_cycles         = 263
num_cycles_per_mul = 5.26
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl alt --input large --stats
num_cycles         = 931
num_cycles_per_mul = 18.62
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl alt --input large_bothneg --stats
num_cycles         = 931
num_cycles_per_mul = 18.62
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl alt --input large_oneneg --stats
num_cycles         = 926
num_cycles_per_mul = 18.52
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl alt --input small_bothneg --stats
num_cycles         = 1550
num_cycles_per_mul = 31.00
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl alt --input small_oneneg --stats
num_cycles         = 250
num_cycles_per_mul = 5.00
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl alt --input mask_lower --stats
num_cycles         = 512
num_cycles_per_mul = 10.24
ECE4750: ~/.../sim/build %
```

Figure 8: the program displaying some information about the number of cycles for the alternative design for small input datasets, large input datasets, large input datasets with either both or one number being negative, small input datasets with either both or one number being negative , and masked lower bits.

```
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input small --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input large --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl alt --input small --stats
num_cycles         = 263
num_cycles_per_mul = 5.26
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl alt --input large --stats
num_cycles         = 931
num_cycles_per_mul = 18.62
ECE4750: ~/.../sim/build %
```

Figure 9: the program displaying some information about the number of cycles for both the baseline and alternative designs for both small and large input datasets.

```
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input small --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input large --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input large_bothneg --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input large_oneneg --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input small_bothneg --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input small_oneneg --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build % ../lab1_imul/imul-sim --impl base --input mask_lower --stats
num_cycles         = 1756
num_cycles_per_mul = 35.12
ECE4750: ~/.../sim/build %
```

Figure 10: the program displaying the number of cycles for the baseline design for small and large input datasets, large input datasets with either both or one negative number , small inputs datasets with either both or one negative number, and masked lower bits.

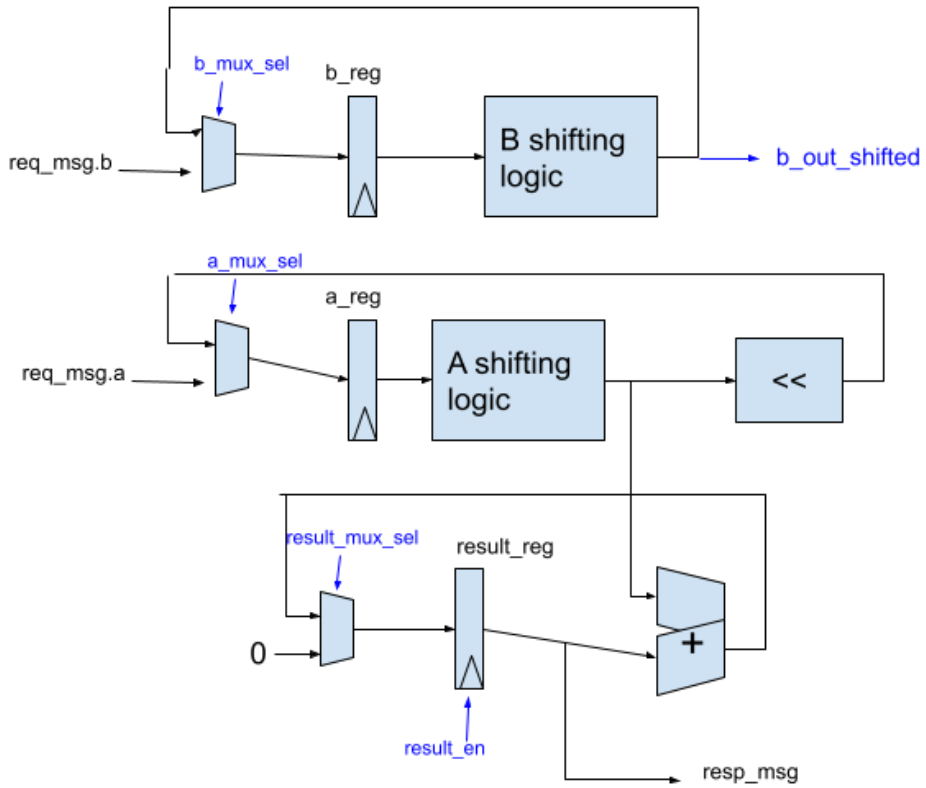


Figure 11: datapath diagram for the alternative design